



DECLARATION OF GORDON MACPHERSON

I, Gordon MacPherson, am over twenty-one (21) years of age. I have never been convicted of a felony, and I am fully competent to make this declaration. I declare the following to be true to the best of my knowledge, information and belief:

1. I am Director Board Governance & IP Operations of The Institute of Electrical and Electronics Engineers, Incorporated (“IEEE”).
2. IEEE is a neutral third party in this dispute.
3. I am not being compensated for this declaration and IEEE is only being reimbursed for the cost of the article I am certifying.
4. Among my responsibilities as Director Board Governance & IP Operations, I act as a custodian of certain records for IEEE.
5. I make this declaration based on my personal knowledge and information contained in the business records of IEEE.
6. As part of its ordinary course of business, IEEE publishes and makes available technical articles and standards. These publications are made available for public download through the IEEE digital library, IEEE Xplore.
7. It is the regular practice of IEEE to publish articles and other writings including article abstracts and make them available to the public through IEEE Xplore. IEEE maintains copies of publications in the ordinary course of its regularly conducted activities.
8. The article below has been attached as Exhibit A to this declaration:

A.	G.H.A. Aty, et al; “High-speed, area-efficient FPGA-based floating-point multiplier”, Proceedings of the 12th IEEE International Conference on Fuzzy Systems, December 11, 2003.
----	--

9. I obtained a copy of Exhibit A through IEEE Xplore, where it is maintained in the ordinary course of IEEE’s business. Exhibit A is a true and correct copy of the Exhibit, as it existed on or about October 13, 2021.
10. The article and abstract from IEEE Xplore shows the date of publication. IEEE Xplore populates this information using the metadata associated with the publication.

Google Exhibit 1093
Google v. Singular

11. G.H.A. Aty, et al; "High-speed, area-efficient FPGA-based floating-point multiplier" was published in the Proceedings of the 12th IEEE International Conference on Fuzzy Systems. Tthe 12th IEEE International Conference on Fuzzy Systems was held on December 11, 2003. Copies of the conference proceedings were made available no later than the date of the conference. The article is currently available for public download from the IEEE digital library, IEEE Xplore.

12. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further that these statements were made with the knowledge that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. § 1001.

I declare under penalty of perjury that the foregoing statements are true and correct.

Executed on: 10/15/2021

DocuSigned by:
Gordon Macpherson
E768DB210F4E4EF...

EXHIBIT A



All



ADVANCED SEARCH

Conferences > Proceedings of the 12th IEEE ...

High-speed, area-efficient FPGA-based floating-point multiplier

Publisher: IEEE

Cite This

PDF

<< Results | Next >

G.H.A. Aty ; A.I. Hussein ; I.S. Ashour ; M. Mones All Authors

3
Paper
Citations246
Full
Text Views

Alerts

Manage Content

Alerts

Add to Citation

Alerts

More Like This

Field programmable gate arrays and floating point arithmetic
IEEE Transactions on Very Large Scale Integration (VLSI) Systems
Published: 1994

Implementation of ANN Training Module on Field Programmable Gate Arrays
2019 Innovations in Intelligent Systems and Applications Conference (ASYU)
Published: 2019

Show More

Abstract



Downl

PDF

Document Sections

1. Introduction
2. Floating-Point Format Representation
3. Floating Point Multiplication
4. Results
5. Conclusion

Abstract:In this paper, a floating-point multiplier with high speed and area efficient is presented. The multiplier is designed, optimized, and implemented on an FPGA based system... **View more**

► Metadata

Abstract:

In this paper, a floating-point multiplier with high speed and area efficient is presented. The multiplier is designed, optimized, and implemented on an FPGA based system. A comparison between the results of the proposed design and a previously reported one is provided. The effect of rounding on the area, speed, and accuracy for three different configurations is examined.

Published in: Proceedings of the 12th IEEE International Conference on Fuzzy Systems (Cat. No.03CH37442)

Date of Conference: 11-11 Dec. 2003 **INSPEC Accession Number:** 8024614

Date Added to IEEE Xplore: 19 April 2004 **DOI:** 10.1109/ICM.2003.237828

Publisher: IEEE

Print ISBN:977-05-2010-1

Authors

Figures

References

Citations

Keywords

SECTION 1.

Introduction

The floating point arithmetic was virtually impossible to be implemented in older generation of FPGAs due to the limited density and speed. In addition, mapping difficulties occurred due to the inherent complexity of floating point arithmetic. Recently, the density and speed of FPGA are increased, so it's easy to implement floating point arithmetic on it. With the introduction of high-level languages such as VHDL, rapid prototyping of floating point units has become feasible. Simulation and synthesis tools at a higher-level design aid the designer for a more controllable and maintainable product. Although low-level design specifications were alternately possible, the strategy used in the work presented here is to specify every aspect of the design in VHDL and rely on automated synthesis to generate the FPGA mapping.

Image and digital signal processing applications require high calculation throughput [1]– [2] [3]. The floating-point multiplier, presented here, is implemented for real-time signal processing on FPGA, which include a 2-D fast Fourier transform (FFT) and a systolic array implementation of an FIR filter. Such signal processing techniques necessitate a large dynamic range of numbers. The use of floating point helps to alleviate the underflow and overflow problems often seen in fixed-point formats. This paper examines the implementations of floating-point multiplier using four floating-point formats similar to the IEEE 754 standard [2]. Floating-point multipliers with 16, 18, 32, and 64 bits have been synthesized for Xilinx 4010 FPGAs [4].

Scalable floating-point multiplier allows manipulating range and precision. Hardware description languages (HDL's) [5] are used for this target. Precision and range can be adjusted by controlling the size of mantissa and exponent fields, respectively. This is done by means of parameter that is passed to HDL at compiling time, this type is called generic. A VHDL generic is a constant value used to parameterize a VHDL design description.

SECTION 2.

Floating-Point Format Representation

The floating-point format that is used in this design is similar to the IEEE 754 standard [6]. For example, single-precision floating-point of IEEE 754 format is shown in Fig. 1.



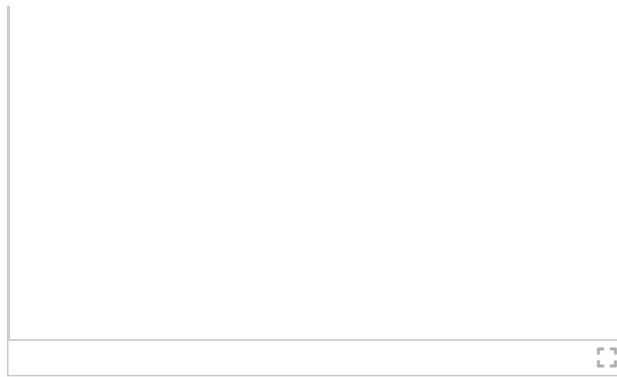
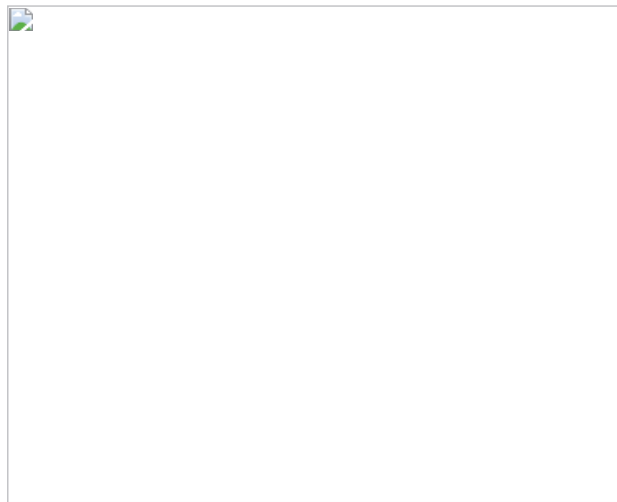


Fig. 1 32 Bit Floating Point Format

The floating-point value (V) is computed by:

$$V = (-1)^S 2^{(e-127)} \quad (1.f)$$

[View Source](#)



As illustrated in Fig. 1, the sign field, s, is bit number 31 and is used to specify the sign of the number. Bits 30 down to 23 are the exponent field. This 8-bit quantity is a signed number represented by using a bias of 127. Bits 22 down to 0 are used to store the binary representation of the floating-point number. The leading one in the mantissa, 1. f, does not appear in the representation, therefore the leading one is implicit. For example, -3.625 (decimal) or -11.101 (binary) is stored in the following way:

$$V = (-1)^1 2^{(128-127)} (1.1101)$$



Where $s = 1$, $e = 128$ (decimal) or 80 (hex), and $f = 680000$ (hex). Therefore, -3.625 is stored as: $C0680000$ (hex). The 18-bit floating-point format is developed in such a way that the dynamic range of the format is large in order to represent very large and small, positive, and negative real numbers accurately.

SECTION 3.

Floating Point Multiplication

Floating-point multiplication is similar to integer multiplication, because floating-point numbers are stored in sign-magnitude form; the multiplier needs only to deal with unsigned integer numbers and normalization. The floating-point multiplier unit is a three-stage pipeline that produces a result on every clock cycle. The bottleneck of this design was the integer multiplier.

There are four different choices for the implementation of the integer multiplier that can be used, namely integer multiplier available in mentor graphic tools with signed or unsigned, array multiplier, parallel multiplier, and series multiplier.

The available unsigned integer multiplier of mentor graphic tools is adopted in this paper.

a. Algorithm

The block diagram of the three-stage, 18-bit floating-point multiplier ($m=10$, $n=7$) is shown in Fig. 2. The algorithm for each stage is as follows:

Stage 1

- The exponents, e_{11} and e_{22} are added and the result along with the carry bit is stored, in an 8-bit register. If the addition of two negative exponents results in a value smaller than the minimum exponent that can be represented, i.e. -63 , underflow occurs. In this case the floating-point number is set to zero. If overflow occurs, the result is set to the maximum number that the format can represent.

- If the floating-point number is not zero, the implied one is concatenated to the left side of the fl1 and f22 terms
- The sign bits are only registered in this stage.
- Integer multiplication of the two 11-bit quantities, 1.f22 and 1.fl1 is performed.

Stage 2

- The output of the integer multiplier will be normalized by shift left until the most significant bit of the mantissa becomes one.
- Adjust the exponent depending on the number of the shifting.
- The sign bits of the two operands are compared. If they are equal to each other the result is assigned a positive sign, but if they differ the result is negative.

Stage 3

- The output of normalization unit will be rounded to return floating-point value into the correct format.
- Remove the bias from the exponent.
- The result-sign, exponent and mantissa fields placed into an 18-bit floating point word.

b. Normalization

The integer multiplier output is normalized by shift left until the most significant bit becomes one. Normalization is an important unit that affects speed and area. Two methods are used to write normalization unit code, either using *for-loop* or *if-statement*; in our design, *if-statement* method is used, which reduces the area by a factor of 2.9 and increases the speed by a factor of 2.6 in comparison to *for-loop* method [7].

C. Rounding

To return the post-processing floating-point value into the correct format, its mantissa bitwidth must be reduced to its original size. However, reduction of bitwidth introduces the need for rounding, because the fraction value needs to be converted into a less precise representation. The IEEE standard specifies four rounding modes:

- Round to zero,
- Round to nearest,
- Round to positive infinity,
- Round to negative infinity,

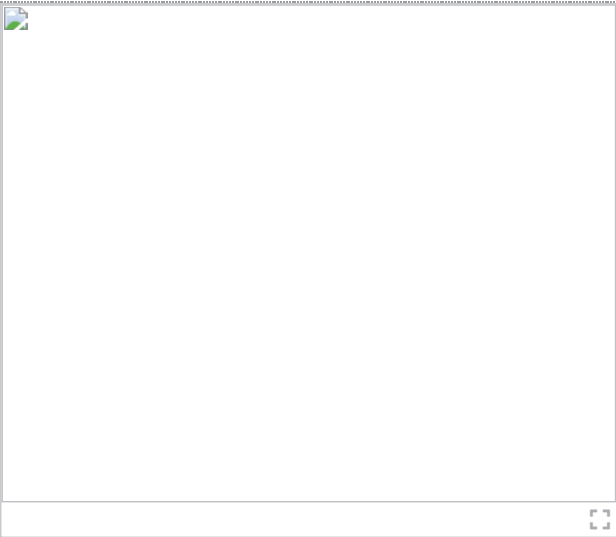
Rounding to zero is the simplest form of rounding to be implemented, because it is represented by simple truncation. For both positive and negative floating point value, removal of low-order bits from the fraction part of the mantissa can only reduce the overall magnitude of the number. However, rounding to zero can introduce significant rounding error in some instance, which

can be avoided by rounding to nearest. For this reason, rounding to nearest is adopted in this paper.

SECTION 4.
Results

The results of three different configurations are presented in Table 1. The results are given for the design of single-precision pipeline floating-point multiplier (32 bit, $m=23$, $n=8$). In the first configuration, shown in Fig. 2, the rounding is carried out after the normalization. In the second configuration, shown in Fig. 3, the rounding is carried out before normalization. The third configuration which is the modified. method presented in this paper is shown in Fig. 4, where the rounding unit is divided in two halves. The first half is done before the normalization while the second half of rounding is done after the normalization. The three different configurations are simulated, synthesized, and mapped on Xilinx vertex II. The results of the three configurations are given in Table 1.

Table 1 Single-Precision Floating-Point Multiplier on Vertex II (2v1000bg575) with Different Designs (F.G: - Function Generators, F.F: Flip Flops)



From Table 1, it is clear that, the area of the second configuration is reduced by 58.6% and the speed is increased by 13% than their counterparts of the first configuration. Due to the complete truncation in the second configuration before normalization unit, the accuracy decreasing dramatically. In our modified configuration (the third configuration) shown in Fig. 4, an optimum design has been selected. The area is reduced by 47% and speed increased by 12.5% than their counterparts of the first configuration, while the accuracy has been improved compared to the second configuration due to partial truncation before normalization.

For comparison purposes, 16 and 18 bits of pipeline floating—point multipliers have been simulated, synthesized, and mapped on Xilinx 4010 using the first configuration which is also used in recently published paper [7]. The code was optimized and the results are given in Table 2. Table 3 shows results from [7]. Comparing the results given in Table 2 and Table 3, it can be

The Vertex II, an improved version of Xilinx, has an embedded multiplier, which reduces the occupied area, and increases the speed of the design implemented on it. The results of mapping the floating-point multiplier on Xilinx Vertex II (2v1000bg575) is given in Table 4. The comparison between the results in Tables 2 and 4 shows that the area is reduced by 47.7% and 71.5% for sixteen and eighteen bit respectively.

SECTION 5.

Conclusion

A design of a floating-point multiplier with an optimized area and speed is presented. The effect of rounding on the area, speed, and accuracy has been examined. The design have been mapped on Vertex II. For future work, an optimized design and implementation of a floating-point adder/ subtractor and accumulator can be investigated and implemented.

Table 2 Optimized Result for Sixteen and Eighteen Bit Floating Point Multipliers on 4010. (F.G: Function Generators, F.F: Flip Flops)

This image shows a blank white page. In the top-left corner, there is a small icon representing a document or folder. In the bottom-right corner, there is a small black square, likely a cursor or a selection tool. The rest of the page is completely empty.

Table 3 Results Recently Published [7] for Sixteen and Eighteen Bit Floating-Point Multiplier on 4010. (F.G: Function Generators, F.F: Flip Flops)



Table 4 Optimized Result for Sixteen and Eighteen Bit Floating Point Multiplier on Vertex Ii. (F.G: Function Generators, F.F: Flip Flops)

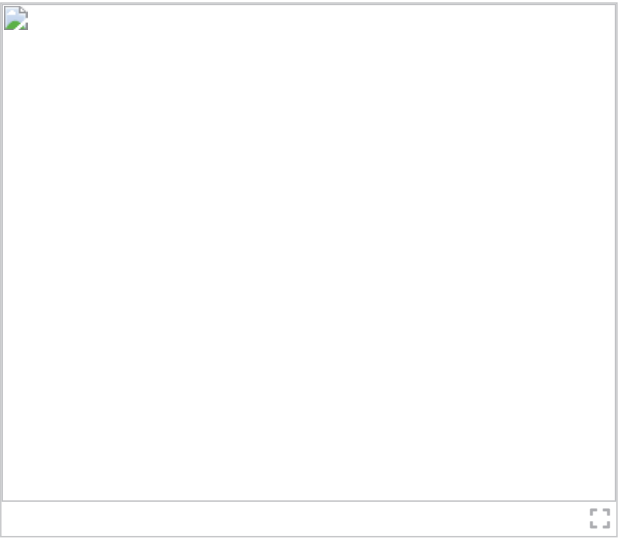
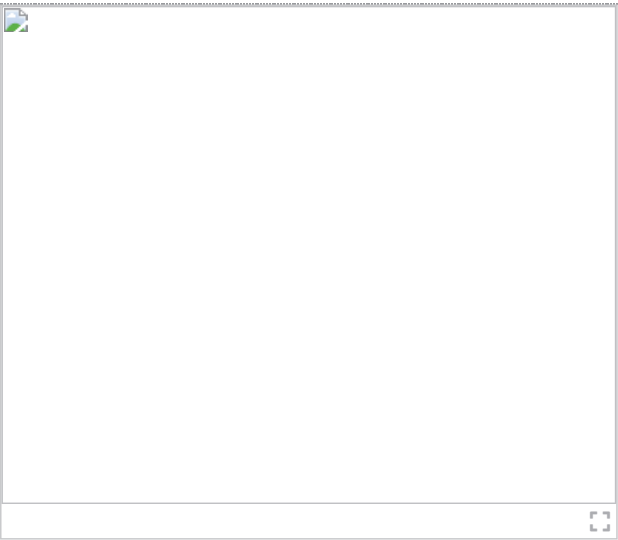


Fig. 2 Shows the first configuration for the design of three-stage, m is the bitwidth of fraction and n is the bitwidth of exponent

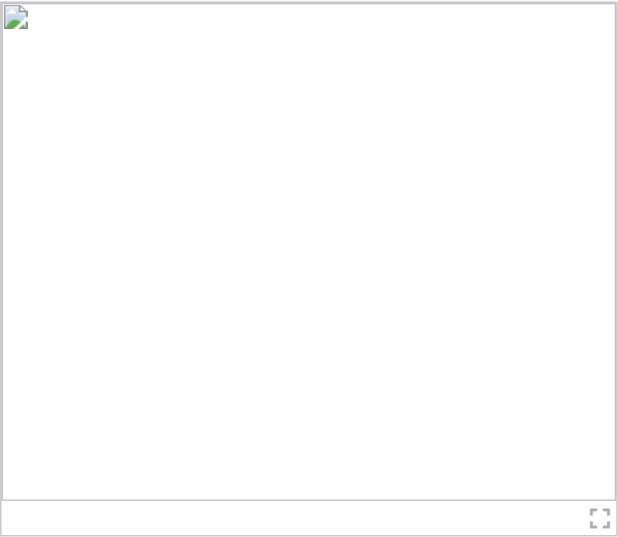


Fig. 3 Shows the second configuration for the design of three-stage, m is the bitwidth of fraction and n is the bitwidth of exponent

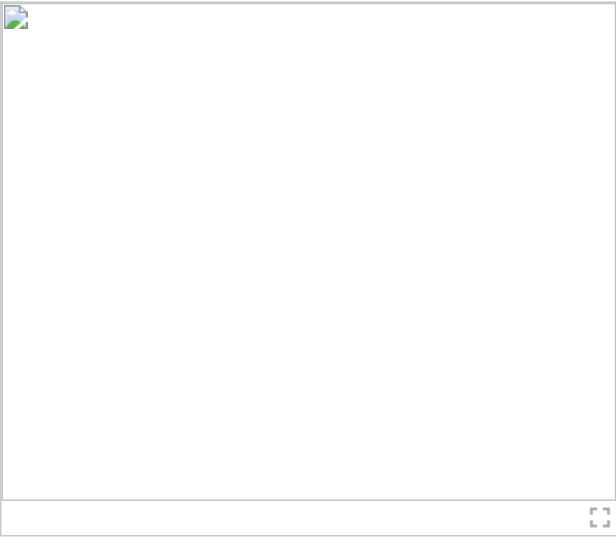


Fig. 4 Shows the third configuration or the design of three-stage, m is the bitwidth of fraction and n is the bitwidth of exponent

Authors	▼
Figures	▼
References	▼
Citations	▼
Keywords	▼
Metrics	▼

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS
VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES
PROFESSION AND EDUCATION
TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333
WORLDWIDE: +1 732 981 0060
CONTACT & SUPPORT

Follow



IEEE Account

- » Change Username/Password
- » Update Address

Purchase Details

- » Payment Options
- » Order History
- » View Purchased Documents

Profile Information

- » Communications Preferences
- » Profession and Education
- » Technical Interests

Need Help?

- » **US & Canada:** +1 800 678 4333
- » **Worldwide:** +1 732 981 0060
- » Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

High-Speed, Area-Efficient FPGA-Based Floating-Point Multiplier

GH. A. Aty
Electronic department
National
Telecommunication
Institute
Cairo, Egypt

Aziza I. Hussein
IEEE Member
Department of Computer &
Systems Engineering
College of Engineering,
Minia University, Minia,
Egypt 61517

I. S. Ashour.
Electronic department,
National
Telecommunication
Institute
Cairo, Egypt

M. Mones
Department of Computer
& Systems Engineering,
College of Engineering,
Minia University, Minia,
Egypt 61517

Abstract: In this paper, a floating-point multiplier with high speed and area efficient is presented. The multiplier is designed, optimized, and implemented on an FPGA based system. A comparison between the results of the proposed design and a previously reported one is provided. The effect of rounding on the area, speed, and accuracy for three different configurations is examined.

1. Introduction

The floating point arithmetic was virtually impossible to be implemented in older generation of FPGAs due to the limited density and speed. In addition, mapping difficulties occurred due to the inherent complexity of floating point arithmetic. Recently, the density and speed of FPGA are increased, so it's easy to implement floating-point arithmetic on it. With the introduction of high-level languages such as VHDL, rapid prototyping of floating point units has become feasible. Simulation and synthesis tools at a higher-level design aid the designer for a more controllable and maintainable product. Although low-level design specifications were alternately possible, the strategy used in the work presented here is to specify every aspect of the design in VHDL and rely on automated synthesis to generate the FPGA mapping.

Image and digital signal processing applications require high calculation throughput [1]-[3]. The floating-point multiplier, presented here, is implemented for real-time signal processing on FPGA, which include a 2-D fast Fourier transform (FFT) and a systolic array implementation of an FIR filter. Such signal processing techniques necessitate a large dynamic range of numbers. The use of floating point helps to alleviate the underflow and overflow problems often seen in fixed-point formats. This paper examines the implementations of floating-point multiplier using four floating-point formats similar to the IEEE 754 standard [2]. Floating-point multipliers with 16, 18, 32, and 64 bits have been synthesized for Xilinx 4010 FPGAs [4].

Scalable floating-point multiplier allows manipulating range and precision. Hardware description languages (HDL's) [5] are used for this target. Precision and range can be adjusted by controlling the size of mantissa and exponent fields, respectively. This is done by means of parameter that is passed to HDL at compiling time, this type is called generic. A VHDL generic is a constant value used to parameterize a VHDL design description.

2. Floating-Point Format Representation

The floating-point format that is used in this design is similar to the IEEE 754 standard [6]. For example, single-precision floating-point of IEEE 754 format is shown in Fig. 1.



Bit #:31 30 ... 23 22 ... 0

Fig. 1 32 Bit Floating Point Format.

The floating-point value (V) is computed by:

$$V = (-1)^s 2^{(e-127)} (1.f)$$

As illustrated in Fig. 1, the sign field, s, is bit number 31 and is used to specify the sign of the number. Bits 30 down to 23 are the exponent field. This 8-bit quantity is a signed number represented by using a bias of 127. Bits 22 down to 0 are used to store the binary representation of the floating-point number. The leading one in the mantissa, 1.f, does not appear in the representation, therefore the leading one is implicit. For example, -3.625 (decimal) or -11.101 (binary) is stored in the following way:

$$V = (-1)^1 2^{(128-127)} (1.1101)$$

Where s=1, e=128 (decimal) or 80 (hex), and f=680000 (hex). Therefore, -3.625 is stored as: C0680000 (hex). The 18-bit floating-point format is developed in such a way that the dynamic range of the format is large in order to

represent very large and small, positive, and negative real numbers accurately.

3. Floating Point Multiplication

Floating-point multiplication is similar to integer multiplication, because floating-point numbers are stored in sign-magnitude form; the multiplier needs only to deal with unsigned integer numbers and normalization. The floating-point multiplier unit is a three-stage pipeline that produces a result on every clock cycle. The bottleneck of this design was the integer multiplier.

There are four different choices for the implementation of the integer multiplier that can be used, namely integer multiplier available in mentor graphic tools with signed or unsigned, array multiplier, parallel multiplier, and series multiplier. The available unsigned integer multiplier of mentor graphic tools is adopted in this paper.

a. Algorithm

The block diagram of the three-stage, 18-bit floating-point multiplier ($m=10$, $n=7$) is shown in Fig. 2. The algorithm for each stage is as follows:

Stage 1:

- The exponents, $e11$ and $e22$ are added and the result along with the carry bit is stored in an 8-bit register. If the addition of two negative exponents results in a value smaller than the minimum exponent that can be represented, i.e. -63, underflow occurs. In this case the floating-point number is set to zero. If overflow occurs, the result is set to the maximum number that the format can represent.
- If the floating-point number is not zero, the implied one is concatenated to the left side of the $f11$ and $f22$ terms
- The sign bits are only registered in this stage.
- Integer multiplication of the two 11-bit quantities, i.e. $f22$ and $f11$ is performed.

Stage 2:

- The output of the integer multiplier will be normalized by shift left until the most significant bit of the mantissa becomes one.
- Adjust the exponent depending on the number of the shifting.

- The sign bits of the two operands are compared. If they are equal to each other the result is assigned a positive sign, but if they differ the result is negative.

Stage 3:

- The output of normalization unit will be rounded to return floating-point value into the correct format.
- Remove the bias from the exponent.
- The result-sign, exponent and mantissa fields placed into an 18-bit floating point word.

b. Normalization

The integer multiplier output is normalized by shift left until the most significant bit becomes one. Normalization is an important unit that affects speed and area. Two methods are used to write normalization unit code, either using *for-loop* or *if-statement*; in our design, *if-statement* method is used, which reduces the area by a factor of 2.9 and increases the speed by a factor of 2.6 in comparison to *for-loop* method [7].

c. Rounding

To return the post-processing floating-point value into the correct format, its mantissa bitwidth must be reduced to its original size. However, reduction of bitwidth introduces the need for rounding, because the fraction value needs to be converted into a less precise representation. The IEEE standard specifies four rounding modes:

- Round to zero,
- Round to nearest,
- Round to positive infinity,
- Round to negative infinity,

Rounding to zero is the simplest form of rounding to be implemented, because it is represented by simple truncation. For both positive and negative floating point value, removal of low-order bits from the fraction part of the mantissa can only reduce the overall magnitude of the number. However, rounding to zero can introduce significant rounding error in some instance, which can be avoided by rounding to nearest. For this reason, rounding to nearest is adopted in this paper.

4. Results

The results of three different configurations are presented in Table 1. The results are given for the design of single-precision pipeline floating-point multiplier (32 bit $m=23$, $n=8$). In the first configuration, shown in Fig. 2, the rounding is carried out

after the normalization. In the second configuration, shown in Fig. 3, the rounding is carried out before normalization. The third configuration which is the modified method presented in this paper is shown in Fig. 4, where the rounding unit is divided in two halves. The first half is done before the normalization while the second half of rounding is done after the normalization. The three different configurations are simulated, synthesized, and mapped on Xilinx vertex II. The results of the three configurations are given in Table 1.

Table 1 Single-precision floating-point multiplier on vertex II (2v1000bg575) with different designs (F.G: Function Generators, F.F: Flip Flops).

Design	F.G	F.F	Speed	Slices
First	1119	117	39.9MHZ	560
Second	463	35	45.2MHZ	232
Third	591	34	44.9MHZ	296

From Table 1, it is clear that, the area of the second configuration is reduced by 58.6% and the speed is increased by 13% than their counterparts of the first configuration. Due to the complete truncation in the second configuration before normalization unit, the accuracy decreasing dramatically. In our modified configuration (the third configuration) shown in Fig. 4, an optimum design has been selected. The area is reduced by 47% and speed increased by 12.5% than their counterparts of the first configuration, while the accuracy has been improved compared to the second configuration due to partial truncation before normalization.

For comparison purposes, 16 and 18 bits of pipeline floating-point multipliers have been simulated, synthesized, and mapped on Xilinx 4010 using the first configuration which is also used in recently published paper [7]. The code was optimized and the results are given in Table 2. Table 3 shows results from [7]. Comparing the results given in Table 2 and Table 3, it can be seen that the area used by the sixteen bit multiplier in our design is reduced by 7 % and the speed is increased by 81.6%, while for the eighteen bit, the area is reduced by 8% and the speed increased by 155 % with respect to results in [7].

The Vertex II, an improved version of Xilinx, has an embedded multiplier, which reduces the occupied area, and increases the speed of the design implemented on it. The

results of mapping the floating-point multiplier on Xilinx Vertex II (2v1000bg575) is given in Table 4. The comparison between the results in Tables 2 and 4 shows that the area is reduced by 47.7% and 71.5% for sixteen and eighteen bit respectively.

5.Conclusion

A design of a floating-point multiplier with an optimized area and speed is presented. The effect of rounding on the area, speed, and accuracy has been examined. The design have been mapped on Vertex II. For future work, an optimized design and implementation of a floating-point adder/subtractor and accumulator can be investigated and implemented.

Table 2 Optimized result for sixteen and eighteen bit floating point multipliers on 4010. (F.G: Function Generators, F.F: Flip Flops).

	Mant	Exp	F.G (800)	F.F (800)	Speed	Stage
16	9	6	232 29%	40 5%	10.9 MHZ	3
18	10	7	290 36%	65 5.88%	12.5 MHZ	3

Table 3 Results recently published [7] for sixteen and eighteen bit floating-point multiplier on 4010. (F.G: Function Generators, F.F: Flip Flops)

	Mant.	Exp.	F.G	F.F	speed	stage
16	9	6	36%	13%	6 MHZ	3
18	10	7	44%	14%	4.9 MHZ	3

Table 4 Optimized result for sixteen and eighteen bit floating point multiplier on vertex II. (F.G: Function Generators, F.F: Flip Flops)

	Mant.	Exp.	F.G	F.F	Speed	Stage
16	9	6	157	58	12 MHZ	3
18	10	7	169	65	12 MHZ	3

References

- [1] J. A. Eldon and C. Robertson, "A Floating Point Format for Signal Processing," Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, pp.717-720, 1982

[2] IEEE Task P754, "A Proposed Standard for Binary Floating-Point Arithmetic," IEEE Computer, Vol. 14, No. 12, pp.51-62, Mar. 1981.

[3] N. Shirazi, Implementation of a 2-D Fast Fourier Transform on an FPGA Based Computing Platform, VPI&SU form on an FPGA Based Computing Platform, VPI&SU

[4] Xilinx, Inc., The Programmable Logic Data Book, San Jose, California, 1993.

[5] K. SKAHILL, "VHDL for Programmable logic," Addison-Wesley Publishing Company, Inc, 2000.

[6] IEEE Standard 754 for Binary Floating Point Arithmetic, IEEE, 1985.

[7] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in Proceedings of IEEE Symposium on FPGAs for custom Computing machines, pp. 155-162, 1995.

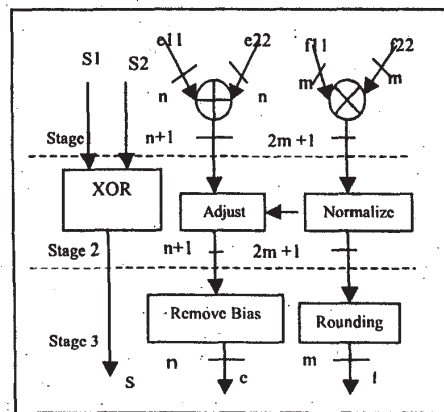


Fig. 2 shows the first configuration for the design of three-stage, m is the bitwidth of fraction and n is the bitwidth of exponent.

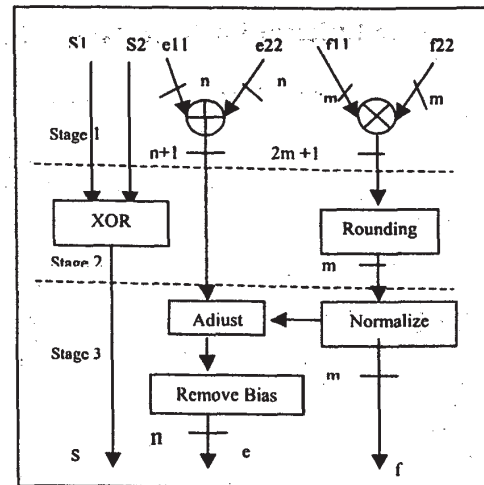


Fig. 3 shows the second configuration for the design of three-stage, m is the bitwidth of fraction and n is the bitwidth of exponent.

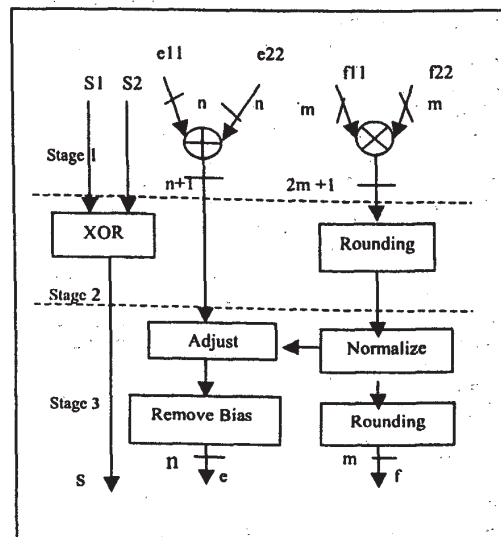


Fig. 4 shows the third configuration or the design of three-stage, m is the bitwidth of fraction and n is the bitwidth of exponent.